DDLBench: Towards a Scalable Benchmarking Infrastructure for Distributed Deep Learning

Matthijs Jansen
SURFsara, University of Amsterdam
Amsterdam, The Netherlands
matthijs.jansen@surfsara.nl

Valeriu Codreanu
SURFsara
Amsterdam, The Netherlands
valeriu.codreanu@surfsara.nl

Ana-Lucia Varbanescu
University of Amsterdam
Amsterdam, The Netherlands
a.l.varbanescu@uva.nl

***Abstract*—Due to its many applications across various fields of research, engineering, and daily life, deep learning has seen a surge in popularity. Therefore, larger and more expressive models have been proposed, with examples like Turing-NLG using as many as 17 billion parameters. Training these very large models becomes increasingly difficult due to the high computational costs and large memory footprint. Therefore, several approaches for distributed training based on data parallelism (e.g., Horovod) and model/pipeline parallelism (e.g., GPipe, PipeDream) have emerged. In this work, we focus on an in-depth comparison of three different parallelism models that address these needs: data, model and pipeline parallelism. To this end, we provide an analytical comparison of the three, both in terms of computation time and memory usage, and introduce DDLBench, a comprehensive (open-source[1], ready-to-use) benchmark suite to quantify these differences in practice. Through in-depth performance analysis and experimentation with various models, datasets, distribution models and hardware systems, we demonstrate that DDLBench can accurately *quantify* the capability of a given system to perform distributed deep learning (DDL). By comparing our analytical models with the benchmarking results, we show how the performance of real-life implementations diverges from these analytical models, thus requiring benchmarking to capture the in-depth complexity of the frameworks themselves.**

I. Introduction

Deep learning (DL) has seen rapid progress in recent years, reaching milestones such as AlphaGo beating the Go world champion [1], and the emergence of deep fakes [2] in 2016. This growth can be attributed to four factors: firstly, the creation of more complex and efficient deep neural networks (DNNs), which can be used for diverse fields of research such as image classification [3] and natural language processing [4]. Secondly, the availability of large and diverse datasets which can be used to train neural networks on, such as both ImageNet [5] and MNIST [6]. Thirdly, the introduction of user-friendly deep learning frameworks such as TensorFlow [7], Keras [8] and PyTorch [9] which opens up the field to researchers without a deep learning background. Finally, the increase in compute power of both CPUs and GPUs has made it possible to train larger models, on more data, in a smaller amount of time [10].

On the one hand, these developments have simplified the field, as deep learning applications can be created in minutes. On the other hand, creating a well-performing application, both in terms of accuracy and training time, is more challenging than ever, due to the abundance of options to choose from in terms of frameworks, neural networks, and hardware. It is often not possible to make an optimal choice for all these parameters without prior information, because the algorithms and techniques involved are of high complexity, especially in a distributed setting. Moreover, with the size-explosion of the models, there are virtually no single-node machines that can efficiently train such models. Instead, models are trained on distributed machines, where the combined memory capacity and computation capability of multiple nodes can be leveraged for efficient training. This adds a new layer of complexity to the already-challenging choices that need to be made.

There are two basic ways towards making a more informed choice: theoretical or empirical. The former, based on the theoretical analysis of models, is captured in frameworks like [11] [12] [13] [14] [15]. These analytical approaches are quick to evaluate, but cannot capture many of the implementation details that often have a significant performance impact (as we can see in Section IV); moreover, they are difficult to expand (without in-depth literature analysis). Empirical approaches, based on benchmarking, provide the most accurate information, as they execute the actual applications on real hardware. Although the benchmarks are more expensive to run, with clever parameterization and generalization, they can be quickly extended to support different models and architectures.

In this work, we propose DDLBench, a *generalizable, ready-to-use benchmark suite for distributed deep learning* (DDL), that can be used to gain insight into the training speed of deep learning applications. DDLBench is designed to support diverse models, datasets, distribution models, and hardware configurations. To demonstrate what the benchmark suite is capable of, we use image classification as an example use case. Our current implementation includes datasets of varying sizes and dimensions: MNIST [6], CIFAR-10 [16], ImageNet [5] and a high-dimensional synthetic dataset. Models-wise, we include MobileNetV2 [17], along with several ResNet [18] and VGG [19] models, thus supporting both compute- or communication-focused networks. Finally, as distribution models, we include Horovod [20] to represent data parallelism, along with GPipe [21] and PipeDream [22] for model and pipeline parallelism. DDLBench is accompanied by an analytical framework to analyze the difference between

[1]https://github.com/sara-nl/DDLBench

the expected performance of the distribution models and the performance of the benchmark suite.

A. Related Work

Measuring and analysing the complexity, training time, and memory usage of SGD and several parallelization methods for deep learning have been popular research topics [12] [23] [24]. However, there is little performance data for the relatively new pipeline parallelism methods, as implemented in GPipe and PipeDream, which *are* included in our analysis.

Benchmark suites and infrastructures such as DAWNBench [25], MLPerf [26], Deep500 [27] and HPC AI500 [28] aim for standardization and generalizability, with the goal of providing a fair comparison between deep learning algorithms, frameworks, and applications. They include a variety of benchmarks and metrics focusing not only on GPU performance, but also on CPU, interconnects, and more. However, none of these benchmarks include or compare data parallelism, model parallelism, and state-of-the-art pipeline parallelism. As these pipeline parallelism distribution models have proven to be an improvement on traditional model parallelism [21] [22], the inclusion of these techniques is vital for any generalizable distributed deep learning benchmark suite.

B. Contributions

This work makes the following contributions to the field of distributed deep learning:

- We define a simple analytical framework to theoretically compare the computational performance of distribution models for deep learning (Section II-D).
- We present the design and implementation of DDLBench, a comprehensive benchmark suite for the evaluation of distributed DNNs (Section III).
- We provide an example of in-depth empirical analysis, facilitated by DDLBench, on 3 distributed models, 6 networks, 4 datasets, and two different clusters (Section IV).
- By comparing our analytical models with the benchmarking results, we show how the performance of real-life implementations diverges from these analytical models, thus requiring benchmarking to capture the in-depth complexity of the frameworks themselves.

Our results indicate data parallelism with Horovod performs as expected, with speedups of up to 4.2 on 4 workers on a single node over a sequential baseline, while pipeline parallelism frameworks unexpectedly underperform: GPipe reaches only up to 3.0 speedup, and PipeDream reaches a modest 1.4 speedup. Both frameworks suffer from inefficient implementations. Our analysis ultimately shows that DDLBench can accurately uncover the complex relation between model training and communication overhead introduced by distribution models together with the changing dynamics when increasing the number of workers.

II. Distributed deep learning

A DNN typically consists of multiple layers: one input layer, one output layer, and one or more hidden layers in between. These layers come in different sizes, shapes and types, such as fully-connected, convolutional, or recurrent. The most popular algorithm to train these models is stochastic gradient descend (SGD) [29], which we use for all benchmarks.

Parallelizing DNN training is complex because the computational workload of training a model has to be divided over multiple workers, thus introducing non-trivial communication patterns. There exist three methods to implement parallel DNN training: data-, model-, and pipeline-parallelism.

A. Data Parallelism

With data parallelism, each worker has a local copy of the full model and trains on a part of the data, either by partitioning or random sampling, thereby reducing the total training time. After a period of local training, the workers exchange their local gradients to calculate the global gradients, which approximate the gradients computed using a sequential approach. The performance of the communication phase is critical in achieving good training time scalability with multiple workers, as the compute workload is equally divided over all workers.

B. Model Parallelism

The opposite of data parallelism is model parallelism, which partitions the model over all workers, so all workers participate in the forward and backward pass of each data element (see Figure 1). The worker processing the first layers uses the input batch of data to train its part of the model on, and then sends the activations of its final layer to the next worker. This process is repeated until the activations have been propagated through the whole model, and is then reversed for the backward pass. As there is a data dependency between workers for both the forward and the backward pass, model parallelism leads to sequential training. The communication overhead of sending activations and gradients is added to this, *resulting in a slowdown with an increasing number of workers*. However, model parallelism supports much larger models compared to data parallelism, because the memory usage per worker is reduced by partitioning over all workers [23]; this reduction allows for larger batch sizes per worker.

C. Pipeline Parallelism

The problem with model parallelism is that, on average, at a given time, less than one worker performs a compute task (Figure 1). Pipeline parallelism provides a direct improvement on model parallelism by introducing more inter- and intra-batch concurrency via data parallelism. The most prominent pipeline distribution models for deep learning models are GPipe [21] and PipeDream [22], both using input pipelining.

GPipe: The input pipelining that GPipe deploys is visualised in Figure 2. GPipe splits each batch into micro-batches, that can not only be processed in parallel, but can also be used to overlap communication with compute tasks such as gradient checkpointing. Gradient checkpointing is implemented in torchgpipe [30], a a GPipe implementation in PyTorch [9], to reduce memory usage and so allowing

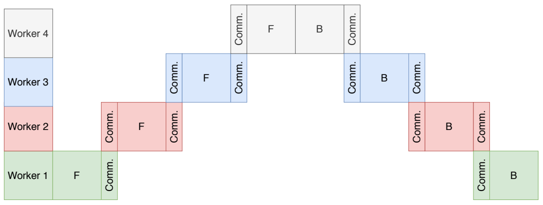


Fig. 1: Execution pipeline of model parallelism.

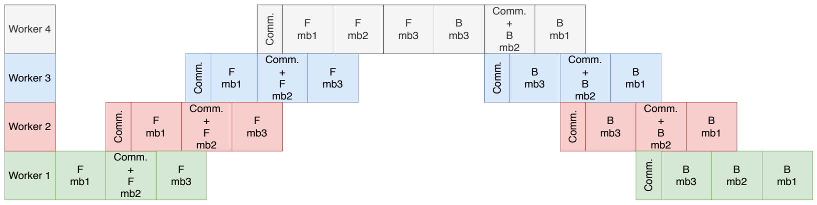


Fig. 2: Execution pipeline of GPipe with 3 micro-batches at a time.

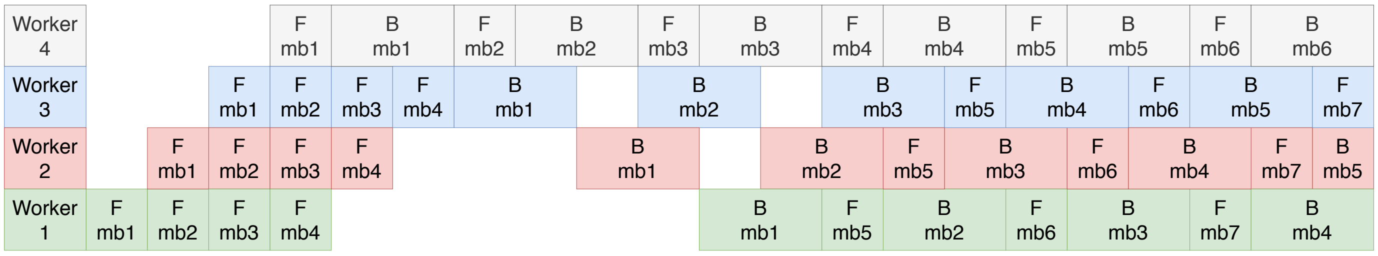


Fig. 3: Execution pipeline of PipeDream with 4 micro-batches at a time where the backward passes takes double the amount of time of the forward pass. The communication overhead is not displayed as it can be overlapped with computations after the first set of micro-batches.

for even larger neural networks to be used. Only a small part of the activations are stored during the forward pass, with the remaining activations being recomputed during the backpropagation while the worker waits for communication to complete. Although the effectiveness of the overlapping depends on how long the communication takes compared to the compute tasks, it still results in much better scaling of the training time with the number of workers compared to model parallelism. This method does not decrease the model's accuracy as long as the micro-batch size does not change, and a synchronization phase is added after the forward and backward pass of each set of micro-batches to update the weights of the model. Only the communication that can not be overlapped with computation is shown in Figure 2.

PipeDream: Where GPipe only uses intra-batch parallelism, PipeDream adds inter-batch parallelism for even faster training. This is achieved by removing the synchronization phase of GPipe after the processing of a set of micro-batches, thereby filling up the execution pipeline of each GPU completely after a startup phase (Figure 3). Removing the synchronization phase does introduce staleness, as micro-batches are trained on old weights. However, the overall time-to-accuracy does not suffer compared to GPipe, because the inter-batch parallelism makes training significantly faster.

Furthermore, PipeDream uses hybrid parallelism [31] [32] [22]: While model partitions interact with one another using pipeline parallelism, each partition can be trained on by multiple workers in parallel using data parallelism. This allows for complex partition configurations which can minimize the communication between workers and so maximise compute efficiency, resulting in lower training times compared to regular pipeline parallelism. As PipeDream does not use gradient recomputation, the number of in-flight micro-batches is quickly limited by the memory capacity of the workers as all activations of all micro-batches needs to be stored.

D. Analytical Modeling

To formalize the differences between these distribution models, we define analytical models for their training time per batch, as the training time per batch is relatively constant over the training process, and therefore can be used to estimate the total training time. The models are presented in Equations 1, 2 and 3 for Horovod, GPipe and PipeDream, respectively. These models are based on previous work on Horovod [20], GPipe [21] [30] and PipeDream [33] [22]. The models will be compared to the empirical benchmark results in Section IV to reason about the predictability of the empirical results, and so the usefulness of the benchmark suite over analytical models.

For all our models, T_{seq} is the training time in sequential execution, W is the number of workers, b is the batch size, and A and G are the sizes of the entire model activation and gradient memory, respectively. The time needed to send data from worker i to worker j is measured in terms of latency $L_{i,j}$ and bandwidth $BW_{i,j}$. For pipeline parallelism, each worker has a copy of a different part of the neural network, and so a different activation and gradient memory, hence the notation A_i and G_i. Finally, $\#mb$ represents the number of micro-batches used per batch.

Horovod: Horovod uses the allreduce operation to exchange the gradients of the loss between workers. It is very difficult to determine beforehand what type of allreduce operation is going to be used at runtime as it depends on the communication backend (NCCL, GLOO, MPI) and the hardware topology. Following [20], we model a ring-allreduce algorithm as an example (Equation 1), although the model can be adapted to any other algorithm as well.

$$\begin{aligned} T_{horovod} = & \frac{T_{seq}}{W} + 2(W-1)\cdot \\ & \max_{i=1}^{W}\left(L_{i,i+1} + \frac{\min(G, th)}{W \cdot BW_{i,i+1}}\right) \end{aligned} \tag{1}$$

The backpropagation is performed on the neural network layer by layer, so one allreduce operation can be performed per layer as well. Using this technique, communication is overlapped with computation as much as possible; however, due to the large amount of allreduce operations being performed, it may be slower than using a single allreduce operation after the backpropagation has finished. Horovod uses Tensor Fusion, performing an allreduce operation once either th MB of gradients (the threshold) have been accumulated during the backpropagation, or x milliseconds have passed. This results in an ordering of allreduce operations at runtime, which cannot be determined beforehand; thus, we assume all allreduce operations can be overlapped with computation, except for the final one, which contains th MB of gradients. This is reflected in Equation 1.

GPipe: GPipe relies on the overlapping of communication with computation for good performance, similar to Horovod (Figure 2). The behaviour of GPipe is modelled in Equation 2. For model parallelism based approaches, typically a much smaller amount of activations and gradients are communicated between workers compared to data parallelism based approaches, although this depends on the type of model used and how the model is divided over the workers. Thus, we assume all communication after the first micro-batch can be overlapped with computation. Model parallelism is implicitly modeled in Equation 2 with one micro-batch.

$$\begin{aligned} T_{gpipe} = & T_{seq} \cdot \frac{W + \#mb - 1}{W * \#mb} + \\ & \sum_{i=1}^{W-1} (L_{i,i+1} + \frac{A_i}{BW_{i,i+1}}) + \\ & \sum_{i=2}^{W} (L_{i,i-1} + \frac{G_i}{BW_{i,i-1}}) \end{aligned} \tag{2}$$

PipeDream: PipeDream is able to fill the execution pipeline of all workers after a startup phase, and can overlap communication with computation just like GPipe. This requires an optimal partition configuration to be used, which can be obtained using PipeDream's built-in profiler. PipeDream's performance is modeled in Equation 3. We assume the time spent in the startup phase is negligible, which is true given enough batches in the dataset.

$$T_{pipedream} = \frac{T_{seq}}{W} \tag{3}$$

In practice, the performance of deep learning applications non-trivially depends on many factors such as the choice of neural network, dataset, distribution model and hardware; factors that are all covered in DDLBench. So, the only way to further increase the accuracy of the analytical models is to perform more benchmarking to calibrate the models [34]. Despite benchmarking being expensive in terms of time and resources, it can also lead to deeper insight into the actual execution of the DL training, thus allowing us to improve on these simplified models.

The analytical models of training time can be used as a basis for memory usage prediction as memory usage depends on the size of the activation and gradient memory per worker. This in turn depends on the number of model partitions, so with the same neural network, data parallelism has the highest memory usage as each worker has a copy of the full model while model parallelism and GPipe have the lowest memory usage as the model is evenly divided over all workers. As PipeDream uses hybrid parallelism, the number of model partitions may very between one and the number of workers, causing the memory usage to fluctuate as well. Besides the number of model partitions, factors such as batch size also influence the memory usage (Section IV-A).

III. Benchmark design and implementation

A. Requirements

Benchmarking Dimensions: The creation, execution, and performance of a distributed deep learning application comes with five degrees of freedom: dataset, neural network architecture, distribution model, deep learning framework and hardware system.

The main goal of DDLBench is to analyze the performance of *all state-of-the-art distribution models* (i.e., data, model, and pipeline parallelism), for **neural networks with different characteristics** (i.e., communication- or computation-intensive), and for **various datasets** (i.e., with different sizes and resolutions). The performance analysis has to be possible for different hardware architectures, ranging from single to multiple nodes, using several GPUs per node.

Metrics: Three of the most important performance metrics for deep learning training are accuracy, training time and memory usage. In this work, *we focus specifically on training time and memory usage*, because accuracy not only depends on the quality of the data and the type of model, but also on hyperparameters like learning rate, momentum and weight decay, making it an application-specific optimization instead of a generalizable optimization which is the case for training time and memory usage. However, DDLBench can be extended to cover more metrics if needed, as discussed in Section V.

B. Design and Implementation

To design a benchmark that meets the requirements presented in Section III-A, we must further select *representative* points in the four dimensions of interest: distribution models, frameworks, neural networks, and datasets.

Distribution Models: The benchmarking suite should include at least one distribution model of each type (i.e., data, model, and pipeline parallelism), to allow us investigate which parallelization method is preferred for different applications.

For data parallelism, we select Horovod [20], which implements data parallelism in a synchronous, decentralized manner - shown to outperform alternative communication solutions; Horovod also supports the majority of deep learning frameworks, making it the preferred choice over framework-specific solutions. For pipeline parallelism we use GPipe [21] [30] and PipeDream [22], two advanced pipeline parallelism

TABLE I: Selected datasets for image classification.

Name	*#classes*	*#images*	*Color profile*	*Resolution*
MNIST	10	70000	Grayscale	28 x 28
CIFAR-10	10	60000	RGB	32 x 32
ImageNet-1000	1000	1280000	RGB	224 x 224
Highres	1000	60000	RGB	512 x 512

TABLE II: Batch size configurations.

Dataset	*PyTorch*	*Horovod*	*GPipe (#mb)*	*PipeDream*
MNIST	128	128	3072 (24)	128
CIFAR-10	64	64	2048 (32)	64
ImageNet-1000	32	32	384 (12)	32
Highres	32	32	48 (12)	32

implementations. We also use these for pure model parallelism by reducing the number of micro-batches processed in parallel to one, so without the use of data parallelism techniques to enhance performance [35] [36].

Frameworks: For implementation purposes, we select PyTorch [9] as the first DL framework for our benchmark suite, because it supports all three distribution models.

Datasets: In its current stage, DDLBench focuses on image classification problems as an example. Thus, our benchmark suite must include datasets with diverse numbers of images and image dimensions. The number of images influences the total training time, because the training process has to be executed for more iterations. The image dimensions influence the size of the input layer and can influence the activation and gradient memory used throughout the network, and thus impact both the compute and communication time: the size of activations and gradients computed per layer and communicated between layers grows with the image dimensions.

To this end, we select three popular [37] datasets for image classification: MNIST [6], CIFAR-10 [16], and ImageNet [5]. We add a fourth, synthetic dataset with high-resolution images, called Highres, to analyze the effect of models with a large memory footprint on training time (Table I).

Neural Networks: Neural networks differ in the type and amount of computation needed due to the total size of the model and the type of layers included in the model and differ in the amount of communication needed due to the number of activations and gradients connecting each layer. We need a selection of neural networks which display diverse amounts of and ratios between computation and communication, to profile all parts of the hardware system. As we have selected datasets related to image classification, we must also select image classification related neural networks, although any type of use case could be used. We select the families of ResNet and VGG models, as they are computation and communication focused, respectively, in a distributed setting [38]. The families contain networks only differing in the number of layers, and so the size of the total network. The current version of DDLBench includes ResNet-18, ResNet-50, ResNet-152, VGG-11 and VGG-16. To this selection, we add MobileNetV2 [17], which uses different types of layers, resulting in different computation and communication patterns.

IV. Evaluation

In this section, we demonstrate how DDLBench can be used for thorough evaluation and performance analysis of distributed DL models, using our diverse combination of networks and datasets.

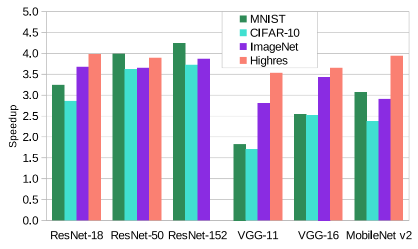
A. Experimental Setup

As we focus on training speed (and not on accuracy), the main performance metric we use is seconds per epoch. We also report speedup, for which we use PyTorch with 1 GPU as baseline. The storage location of the training data is critical for training speed, as storage in a central location could lead to network contention, resulting in fluctuating training times [39]. Although it would be interesting to analyse how the different distribution models perform when directly reading the data from a central storage, it is not the focus of this research; therefore, we write synthetic versions of the datasets (Table I) to the local storage of each node. As there are no other factors which can cause major fluctuations in the training time per epoch, we limit the number of epochs to 3[2], and report the average training time per epoch.

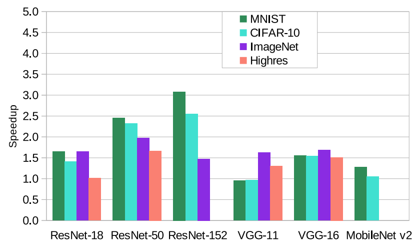
a) Batch Size: The increase in batch size affects memory usage, as it leads to larger activation and gradient memory sizes and more images are loaded into memory at once. This makes choosing a batch size an application-specific optimization rather than a generalizable one, so all our experiments use one batch size per dataset per distribution model. It also affects the convergence rate as larger batch sizes tend to result in lower convergence rates, although this can be compensated by increasing the learning rate [40]. The benchmark suite includes tools to track memory usage per worker. These tools are used to optimize the batch sizes for memory usage (Table II). For Horovod, the batch size per worker is reported. For GPipe, the batch size differs per application based on the number of workers, as with more workers, each worker gets a smaller portion of the model, resulting in less memory usage per worker. We report the batch sizes for 4 workers as torchgpipe [30] currently does not support multi-node setups. For PipeDream the same batch sizes as for the sequential applications are used because of limitations in PipeDream's profiler, which is used to automatically partition the neural network over all workers. Furthermore, determining the optimal batch size for a PipeDream application is extra difficult because a combination of data and pipeline parallelism is used.

b) Hardware: We use two different clusters for our experiments. *Cluster-A* uses NVIDIA Titan RTX GPUs with 24 GB of device memory. Each node has 4 GPUs, two pairs connected with NVLink, and dual-socket Intel Xeon Gold 5118 CPUs with 192 GB of RAM. The nodes are connected via 40 Gbps Ethernet. *Cluster-B* uses the NVIDIA GeForce

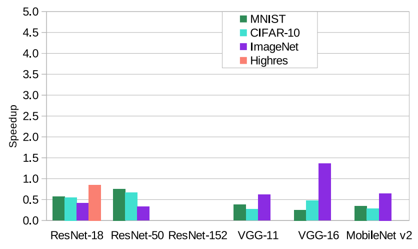
[2]In fact, this is a user-configurable parameter in DDLBench, but we find 3 to provide a good balance between quick experimentation and representative timing information.



(a) Horovod



(b) GPipe



(c) PipeDream

Fig. 4: Training time speedup compared to PyTorch with 1 GPU using 4 GPUs on Cluster-A.

1080Ti with 11 GB of device memory. Each node has 4 GPUs and dual-socket Intel Xeon Bronze 3104 CPUs with 256 GB of RAM. The nodes are connected via 40 Gbps Ethernet as well. Cluster-A has faster GPUs and higher intra-node bandwidth due to NVLink, while both clusters have the same inter-node capabilities for multi-node workloads. All nodes use GCC 8.3.0, CUDA 10.1.243, cuDNN 7.6.5.32, OpenMPI 3.1.4 and NCCL 2.5.6. We use PyTorch 1.5.0 for all experiments except those with PipeDream, which use PyTorch 1.1.0 (due to changes made in the source code of PyTorch by PipeDream). As the currently included distribution models rely on PyTorch for the execution of the code, DDLBench supports any GPU that can be used with PyTorch.

B. *Single-Node Benchmarks*

First, we perform benchmarks using a single node of Cluster-A. All selected datasets, neural networks and distribution models are being used, as shown in Figure 4. Some experiments did not succeed for GPipe and PipeDream due to unexpected behaviour. In our experience, this behavior demonstrates the lack of maturity for pipeline parallelism frameworks, when compared to established, data-parallel ones, such as Horovod.

Horovod: Horovod performs very well, even achieving a super-linear speedup with MNIST and ResNet-152 as each GPU processes only a quarter of the data, and therefore can cache more data. The results in Figure 4a are in line with the performance model in Equation 1: Computation-intensive networks such as ResNet perform better than communication-intensive networks such as VGG due to a reduced communication overhead. The high dimensional dataset Highres performing the best out of all datasets is also as expected as an increase in image resolution results in a more computation-intensive application in this example.

GPipe: GPipe follows the same trends as Horovod: Good scalability for computation-intensive networks and much less performance increase for communication-intensive networks (Figure 4b). The performance is similar to that of [30] and [21]. However, the results are not in line with Equation 2, as GPipe should have much less communication overhead compared to Horovod due to the small amount of activations and gradients being communicated between workers. Furthermore, the average speedup gained by the GPipe benchmarks is much less than that of Horovod (Figure 4). Detailed performance analysis of torchgpipe has shown that the forward pass, as implemented in torchgpipe, has been implemented very inefficiently compared to the backward pass, which is automatically executed by PyTorch autograd, PyTorch's backpropagation engine [34]. For one, the computation and communication is not overlapped during the forward pass, instead these two phases are executed in sequence. This, together with more inefficiencies, explains the gap between the expected performance and the actual performance. For memory usage, GPipe uses much larger batch sizes compared to Horovod, and can train ResNet-152 on ImageNet where Horovod can not due to memory limitations, showing that speedup is not the only import metric to compare these distribution models with.

PipeDream: The PipeDream benchmarks achieve a positive speedup compared to PyTorch with one worker in only a single case. This is surprising, as PipeDream should in theory perform the best out of the three distribution models as its combination of pipeline and data parallelism, together with the lack of synchronization compared to GPipe, should result in an optimal reduction of communication overhead. One cause could be the small batch size compared to GPipe, however this is due to limitations in the PipeDream framework. Detailed performance analysis in [34] shows that PipeDream uses PyTorch's DistributedDataParallel library for asynchronous data parallelism [22]. However, PyTorch does not support asynchronous data parallelism as implemented in PipeDream, completely changing PipeDream's execution pipeline.

C. *Performance Scaling*

Data, model and pipeline parallelism all exhibit strong scaling as either the training data or the model is divided over all workers, reducing the computational workload per worker. As such, the communication overhead eventually becomes the bottleneck for all distribution models. However, because GPipe and PipeDream have less communication between workers, and can overlap it with computation, they should scale better. We performed experiments up to 16 GPUs using the ImageNet dataset and a selection of neural networks to find out if these predictions apply here. GPipe has not been included as it does not support multi-node configurations in PyTorch [30]. We do however include model parallelism by using PipeDream without hybrid parallelism and input pipelining,

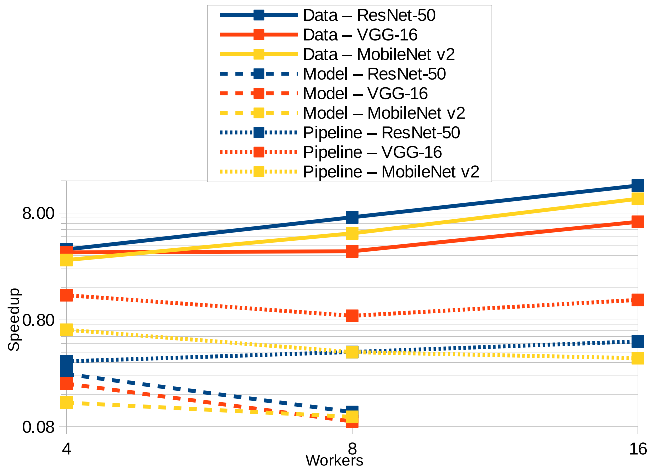


Fig. 5: Training time speedup compared to PyTorch with 1 GPU using 1, 2 and 4 nodes with 4 GPUs each on Cluster-A. Horovod is used for data parallelism and PipeDream for both model and pipeline parallelism.

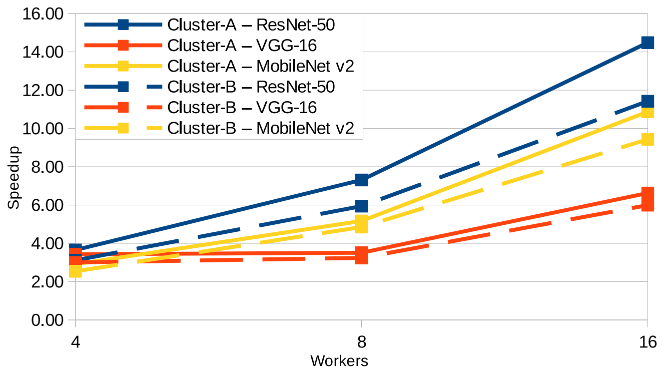


Fig. 6: Training time speedup for Horovod (compared to PyTorch with 1 GPU) using 1, 2 and 4 nodes with 4 GPUs each on Cluster-A and Cluster-B.

to show the difference in performance between model and pipeline parallelism.

Horovod: Horovod again performs very well, achieving a speedup of 14.5 over PyTorch with one worker using ResNet-50 (Figure 5). For VGG-16 and MobileNet v2 it achieves a sub-linear, but constant speedup as well. The difference in performance between ResNet-50 and VGG-16 confirms that the total training time is indeed dominated by communication overhead, as Horovod can no longer successfully overlap communication with computation.

Figure 6 shows that the speedups achieved on Cluster-B are between 5 and 20 percent slower than those achieved on Cluster-A. This is not surprising as the GPUs in Cluster-B are much slower than those in Cluster-A. This, coupled with the fact that both clusters share the same interconnect, also explains why on Cluster-B, the computation-intensive benchmarks scale slightly worse than the communication-intensive benchmarks compared to the same benchmarks on Cluster-A.

PipeDream: Figure 5 clearly shows that pipeline parallelism is an improvement of model parallelism as it performs better for all experiments. Moreover, the speedup of the model parallelism benchmarks decreases when increasing the number of workers, as this type of distribution model lacks concurrency. For this reason we have not performed experiments with model parallelism using more than 8 workers.

Figure 5 also shows that pipeline parallelism does not scale well with an increasing number of workers, if at all, in contrast to what is predicted in Equation 3. Surprisingly, PipeDream performs best with VGG-16. This unexpected behaviour is most likely due to the earlier mentioned difference between the PipeDream's expected execution pipeline [33] [22] and the actual execution pipeline [34]. Surprisingly, our results for PipeDream differ greatly from those in [22], which reports speedups of up to 1.0 for ResNet-50 and up to 5.28 for VGG-16 compared to a data parallel approach, using ImageNet and 16 workers. This has been achieved by manually optimizing the batch sizes and hybrid parallelism configurations, while we used the automatic performance optimization tools from the PipeDream framework due to the goal of DDLBench being generalizability. Moreover, these experiments were performed on a cluster with a much slower interconnect, which benefits pipeline parallelism more than data parallelism due to its reduced communication between workers.

V. Benchmark Limitations and Extensibility

The need for a distributed benchmark suite is clear: the performance of deep learning applications depends on too many factors and is too dynamic to predict using analytical methods. To increase the accuracy of the current analytical models (Equation 1, 2 and 3), *benchmarking is mandatory* to tweak and correctly calibrate such analytical models. With enough performance data, these models can be further enhanced using statistical methods. These accurate performance models could then be used as performance predictors, thus avoiding unnecessary, time- and resource-consuming benchmarking. Finally, such predictors can enable the construction of a distributed deep-learning recommender system [34], which can determine the best distribution model for a given workload *before execution*.

The scope of the current instance of DDLBench is limited to image classification. However, extending DDLBench to other use cases such as natural language processing is easy as most deep learning frameworks support a variety of use cases. In this way, to make a new distributed application for the benchmark suite, one only needs to create a sequential application using a deep learning framework, and extend it with a distribution model.

It is also possible to extend DDLBench to cover more deep learning frameworks, beside PyTorch. In this way, new distribution models that are only supported by a particular framework can be used. Supporting a new framework requires the user to create new sequential applications using that framework's functionalities, and extend it with a distribution model supported by that framework.

To support more distribution models in the benchmark suite, it is important to note that these models come in two types: first, software library-based models like Horovod and GPipe, which are to be used to extend the source code of sequential applications to transform them into distributed applications. Secondly, standalone frameworks like PipeDream, which can be used using a command line interface. Possible new distribution models for DDLBench are Megatron-LM [35] and Deepspeed [41], both based on model parallelism.

The metrics of choice for DDLBench are time per epoch and memory usage. However, for in-depth analysis, more metrics may be needed, such as accuracy, CPU usage, network bandwidth or Valid FLOPS [28]. Most of these metrics are already included in deep learning frameworks (i.e. PyTorch) or hardware drivers (i.e. NVIDIA-smi), and can be easily incorporated into the benchmarking infrastructure.

DDLBench can also be extended to perform automatic performance optimization (which otherwise requires significant manual effort). Distribution models such as GPipe and PipeDream already include automatic model partitioning for pipeline parallelism, but automatic tuning for batch size and number of micro-batches is not yet possible. Such automated optimization enables the benchmark suite to show the maximum performance a distribution model can achieve.

Instead of extending DDLBench, it is also possible to integrate our benchmark suite into a more mature framework, such as DAWNBench [25], MLPerf [26], Deep500 [27] or HPC AI500 [28]. However, such integration poses many technical challenges, as different distribution models do not only require different deep learning frameworks, but also different versions of the same deep learning framework. Furthermore, there are multiple popular deep learning benchmark suites, so which benchmark suite to choose to build upon is unclear. While we support the goal of creating a single benchmark suite to support a large variety of state-of-the-art distribution models, such a benchmark must start from in-depth analysis of user requirements, a detailed blue print for its design, and significant community effort for its implementation. Before such agreements are reached, we believe the grass-roots development of benchmarks for different, specific and novel aspects of deep learning should and will continue.

VI. Conclusion

Deep neural networks grow larger and more complex, especially with the emergence of new applications and massive datasets. In this context, distributed training is a promising avenue for sustainable scaling in terms of model complexity and dataset size. However, understanding how to efficiently use distributed training models is important not only for the time-to-solution, but also for the resource allocation and utilization during training.

To better understand the computational performance of distributed training, we present DDLBench, the first generalizable, ready-to-use benchmark suite, designed to gain insight into the impact of the distribution model on the training speed of deep learning applications. DDLBench is able to profile the training speed and memory usage of DL applications, and supports several datasets, neural networks, distribution models, frameworks and hardware systems. We formulate simple theoretical models to capture the main performance factors in distributed DL training, and use them as a coarse reference for the expected performance. Further, through experimentation, we show that DDLBench can accurately quantify the complex relation between model training and communication overhead introduced by distribution models, together with the changing dynamics when increasing the number of workers. Overall, our results indicate that performance models must capture much better such complex behaviour, and this is not possible with simple, theoretical analytical models. In fact, we advocate benchmarking as a tool for improving analytical models, to the point where their accuracy can be good enough to enable their use as predictors in, for example, choosing the right model for a given application.

The current implementation of DDLBench could be extended by adding automated tuning for batch size and number of micro-batches, to further optimize the performance of each benchmark individually, showing the maximum performance the distribution models can achieve. Additionally, more models, datasets, and networks can be added for increased diversity, and more metrics can be added for in-depth analysis. Finally, the integration with other benchmarks is possible, as long as the novel design dimensions of DDLBench are included, and the software engineering challenges of merging multiple software technologies in one portable benchmark are also solved.

References

[1] F.-Y. Wang, J. J. Zhang, X. Zheng, X. Wang, Y. Yuan, X. Dai, J. Zhang, and L. Yang, "Where does alphago go: From church-turing thesis to alphago thesis and beyond," *IEEE/CAA Journal of Automatica Sinica*, vol. 3, no. 2, pp. 113–120, 2016.

[2] J. Thies, M. Zollhofer, M. Stamminger, C. Theobalt, and M. Nießner, "Face2face: Real-time face capture and reenactment of rgb videos," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2387–2395.

[3] A. Khan, A. Sohail, U. Zahoora, and A. S. Qureshi, "A survey of the recent architectures of deep convolutional neural networks," *arXiv preprint arXiv:1901.06032*, 2019.

[4] L. Dong, S. Xu, and B. Xu, "Speech-transformer: a no-recurrence sequence-to-sequence model for speech recognition," in *2018 IEEE ICASSP*. IEEE, 2018, pp. 5884–5888.

[5] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[6] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[7] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[8] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[9] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[10] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking state-of-the-art deep learning software tools," in *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*. IEEE, 2016, pp. 99–104.

[11] S. Shi, Q. Wang, and X. Chu, "Performance modeling and evaluation of distributed deep learning frameworks on gpus," in *2018 IEEE 16th DASC/PiCom/DataCom/CyberSciTech*. IEEE, 2018, pp. 949–957.

[12] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–43, 2019.

[13] S. Alqahtani and M. Demirbas, "Performance analysis and comparison of distributed machine learning systems," 07 2019.

[14] Z. Li, W. Yan, M. Paolieri, and L. Golubchik, "Throughput prediction of asynchronous sgd in tensorflow," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 76–87. [Online]. Available: https://doi.org/10.1145/3358960.3379141

[15] D. Justus, J. Brennan, S. Bonner, and A. S. McGough, "Predicting the computational cost of deep learning models," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 3873–3882.

[16] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[17] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[19] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[20] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[21] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," in *Advances in Neural Information Processing Systems*, 2019, pp. 103–112.

[22] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 1–15.

[23] S. Alqahtani and M. Demirbas, "Performance analysis and comparison of distributed machine learning systems," *arXiv preprint arXiv:1909.02061*, 2019.

[24] M. Naumov, "Parallel complexity of forward and backward propagation," *arXiv preprint arXiv:1712.06577*, 2017.

[25] C. Coleman, D. Narayanan, D. Kang, T. Zhao, J. Zhang, L. Nardi, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia, "Dawnbench: An end-to-end deep learning benchmark and competition," *Training*, vol. 100, no. 101, p. 102, 2017.

[26] P. Mattson, C. Cheng, C. Coleman, G. Diamos, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf *et al.*, "Mlperf training benchmark," *arXiv preprint arXiv:1910.01500*, 2019.

[27] T. Ben-Nun, M. Besta, S. Huber, A. N. Ziogas, D. Peter, and T. Hoefler, "A modular benchmarking infrastructure for high-performance and reproducible deep learning," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 66–77.

[28] Z. Jiang, L. Wang, X. Xiong, W. Gao, C. Luo, F. Tang, C. Lan, H. Li, and J. Zhan, "Hpc ai500: The methodology, tools, roofline performance models, and metrics for benchmarking hpc ai systems," *arXiv preprint arXiv:2007.00279*, 2020.

[29] A. Trask, *Grokking deep learning*. Manning Publications Co., 2019.

[30] C. Kim, H. Lee, M. Jeong, W. Baek, B. Yoon, I. Kim, S. Lim, and S. Kim, "torchgpipe: On-the-fly pipeline parallelism for training giant models," 2020.

[31] Y. Oyama, N. Maruyama, N. Dryden, E. McCarthy, P. Harrington, J. Balewski, S. Matsuoka, P. Nugent, and B. Van Essen, "The case for strong scaling in deep learning: Training large 3d cnns with hybrid parallelism," *arXiv preprint arXiv:2007.12856*, 2020.

[32] B. Van Essen, H. Kim, R. Pearce, K. Boakye, and B. Chen, "Lbann: Livermore big artificial neural network hpc toolkit," in *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, 2015, pp. 1–6.

[33] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons, "Pipedream: Fast and efficient pipeline parallel dnn training," *arXiv preprint arXiv:1806.03377*, 2018.

[34] M. Jansen, "A performance-based recommender system for distributed dnn training," Master's thesis, University of Amsterdam, Amsterdam, the Netherlands, 8 2020. [Online]. Available: https://github.com/redplanet00/A-performance-based-recommender-system-for-Distributed-DNN-training

[35] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using gpu model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[36] A. Gholami, A. Azad, P. Jin, K. Keutzer, and A. Buluc, "Integrated model, batch, and domain parallelism in training neural networks," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, 2018, pp. 77–86.

[37] "Papers with code," https://paperswithcode.com/task/image-classification, accessed: 2020-04-01.

[38] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, and P. B. Gibbons, "Pipedream: Fast and efficient pipeline parallel DNN training," *CoRR*, vol. abs/1806.03377, 2018. [Online]. Available: http://arxiv.org/abs/1806.03377

[39] C.-C. Yang and G. Cong, "Accelerating data loading in deep neural network training," *arXiv preprint arXiv:1910.01196*, 2019.

[40] E. Hoffer, I. Hubara, and D. Soudry, "Train longer, generalize better: closing the generalization gap in large batch training of neural networks," in *Advances in Neural Information Processing Systems*, 2017, pp. 1731–1741.

[41] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *26th ACM SIGKDD*, 2020, pp. 3505–3506.